|  |
| --- |
|  |
| COMP 3203 Assignment 2 |
| Project Report |
|  |
| **Anthony D’Angelo # 100773125, Nicolas Porter # 100760059** |
| **11/30/2011** |

|  |
| --- |
| This document provides instructions on how to use the project application followed by class diagrams, implementation details, rudimentary test cases and, lastly, example log file output. |

Contents

[Table of Figures 2](#_Toc310426081)

[Table of Tables 3](#_Toc310426082)

[Implementation Details 3](#_Toc310426083)

[JGraphT 3](#_Toc310426084)

[JGraph 4](#_Toc310426085)

[Directed Sensor Antennae Orientation Algorithm 4](#_Toc310426086)

[Performance Comparisons 6](#_Toc310426087)

[Testing 7](#_Toc310426088)

[Logging 12](#_Toc310426089)

## Table of Figures

[Figure 1: As we can see, when the sector angle is 180 degrees and they are aligned near perfectly straight, the algorithm orients the antennae such that they form a strongly connected graph 7](#_Toc310425996)

[Figure 2: The omnidirectional version of the graph shown in Figure 1 8](#_Toc310425997)

[Figure 3: A manually, randomly shaped network using a 180 degree sector angle. Note that it is strongly connected. If we refer to Figure 4 we'll see that this is in fact a strongly connected graph of the omnidirectional counterpart. 8](#_Toc310425998)

[Figure 4: The omnidirectional graph of the network in Figure 3 9](#_Toc310425999)

[Figure 5: When contrasted to Figure 6 shows how a range change for a given sector angle can affect your connectivity. With a difference of 50 units (randomly chosen difference) these two graphs differ in connectivity. 9](#_Toc310426000)

[Figure 6: Refer to Figure 5 10](#_Toc310426001)

[Figure 7: Another manually, randomly shaped omnidirectional graph. In some cases (this being one of them) the graph created by the orientation algorithm when its sector angle is 360 degrees (Figure 8) will have edges in the same spots as the omnidirection version which helps us validate the result of the algorithm. 10](#_Toc310426002)

[Figure 8: Refer to Figure 7 11](#_Toc310426003)

[Figure 9: When viewed with Figure 10 goes to show that the edges in the omnidirectional graph don’t always get mapped to the directed version as was sort of the case in Figure 7 and Figure 8. 11](#_Toc310426004)

[Figure 10: When we compare this graph to the one in Figure 9, although it is still strongly connected, it doesn’t have all of the same edges. This is a direct consequence of the algorithm and happens because the only vertices for which we make a connection to everything within reach are the vertices in the matching whereas the omnidirectional graph connects each vertex to everything it can reach. 12](#_Toc310426005)

## Table of Tables

[Table 1: Useful JGraphT algorithms used on the graph classes in our program 4](#_Toc310426006)

[Table 2: Test 1 7](#_Toc310426007)

[Table 3: Test 2 8](#_Toc310426008)

[Table 4: Test 3 9](#_Toc310426009)

[Table 5: Test 4 10](#_Toc310426010)

[Table 6: Test 5 11](#_Toc310426011)

## Implementation Details

### JGraphT

Our program implements graphs using graph data structures and graph algorithms from the JGraphT library. The *ProximityGraph* class extends from *SimpleGraph*, a JGraphT class which models an undirected graph with no self loops or multiple edges between a pair of vertices. We use this *ProximityGraph* class to represent our omnidirectional graphs. Instances of the *Sensor* class have properties like orientation, position and unique numeric identifiers. This *ProximityGraph’*s constructor requires a set of sensors which is then used as its vertex set which is itself then used to compute the graph’s edges. We store the vertices and edges of the graphs in instances of the standard Java *Set* template classes because they are easy to work with and they do not allow duplicate items. The *ProximityGraph* computes its edge set by creating/adding an edge between every node pair whose Euclidian distance is smaller than the range of the sensors. Note that the range and sector angle of sensors is a class static variable in the *Sensor* class which means that all sensors will have the same range and sector angle (although only directed graphs care about the sector angle). The term “ProximityGraph” was used because the paper describing the orientation algorithm for directed graphs (mentioned below in the “Directed Sensor Antennae Orientation Algorithm” section) used this term which, to us, seemed to describe an omnidirectional graph because the edges are between a given vertex and the vertices in the proximity of the given vertex.

The *TransmissionGraph* class extends the *SimpleDirectedGraph* class (another JGraphT class) which is the same as a simple graph but its edges are directed. Like the *ProximityGraph*, this class' constructor takes in a set of sensors to use as its vertex set which is then used to compute the graph’s edges. This constructor differs from *ProximityGraph’*s though because to compute the edge set it uses the sensor antennae orientations which it also computes (by using the orient antenna algorithm which is described in the “Directed Sensor Antennae Orientation Algorithm” section). The term “TransmissionGraph” was used because the paper describing the orientation algorithm for directed graphs (mentioned below in the “Directed Sensor Antennae Orientation Algorithm” section) used this term which, to us, seemed to describe a directional graph because the creation of edges must now restrict and define to whom a given vertex can transmit.

Implementing our graph data structures by extending JGraphT classes allows us to make use of the JGraphT’s algorithm classes in Table 1.

Table : Useful JGraphT algorithms used on the graph classes in our program

|  |  |
| --- | --- |
| **Class/Algorithm** | **Used to:** |
| **StrongConnectivityInspector** | test a graph for strong connectivity |
| **FloydWarshallShortestPaths** | compute all of the shortest paths from some node to every other node in a graph |
| **BreadthFirstIterator** | graph traversal |
| **KruskalMinimumSpanningTree** | compute the minimum spanning tree of a graph |

### JGraph

To display our graphs we use the JGraph library. This library is used to draw nodes as well as undirected and directed edges on a Swing JComponent. It also supports the dragging of nodes (user relocation of nodes). JGraphT provides a *JGraphAdapter* class, which allows you to use a JGraphT graph (data structure) in JGraph. Since our graph classes inherit from JGraphT classes, we can wrap them in the adapter and display them with minimal effort.

At the Graphical User Interface (GUI) level, our graphs are built using the *GraphFactory* interface whose purpose is to give the GUI a way to build any graph from a set of sensors. We have two implementations of this interface, the *ProximityGraphFactory* and *TransmissionGraphFactory* classes.

To draw the antenna broadcast area of a sensor, we hook into JGraph’s *paint*() method and draw the area (a disk or a disk sector) in the method’s off-screen buffer.

We create a new graph when the set of sensors is modified through the GUI. This ensures that all edges are recomputed accurately.

### Directed Sensor Antennae Orientation Algorithm

The algorithm used for orienting the directed sensors was extracted from section 4 (*Approximating the minimum radius*) from the paper *Communication in Wireless Networks with Directional Antennae* by Ioannis Caragiannis, Christos Kaklamanis, Evangelos Kranakis, Danny Krizanc and Andreas Wiese and then converted into code that simply orients the sensors. The coded algorithm uses the same predefined radius and sector angle for all nodes and does not try to determine an optimum radius or an optimum angle.

The algorithm takes in a set of vertices V and computes the minimum spanning forest on it, but it only uses one tree of the forest to make its computations. This would mean that only one connected subgraph is computed, but by iterating through the connected sets of V we are able to effectively orient all subgraphs regardless of whether or not they are in the first tree we saw. At a high level, the operations we perform are as follows: we compute the minimum spanning tree (MST); we use the MST to determine a matching; any vertex not in the matching is oriented towards its parent; every vertex in every couple in the matching orients itself with its coupled neighbor to cover the maximum area they can and then they make connections to any vertices they can reach (in a strongly connected graph, this last step is when the communication to the leaves is enabled). Below we talk a little more in depth about how these are accomplished.

Computing the MST was quite straightforward thanks to the JGraphT library. There is a class *KruskalMinimumSpanningTree* (KMST) to whose constructor you pass the graph on which to make the MST. We pass in a new *ProximityGraph* created using V and are returned an object of the type KMST. Then we populate a new *SimpleGraph* (which we treat as our MST) with all of the vertices in V and all of the edges in the edge set returned by the KMST object.

The pain of creating a matching was also eased, once again, by the JGraphT library. There is a class *BreadthFirstSearchIterator* (BFSI) to whose constructor is passed the graph on which we want to traverse and the start vertex which is treated as the root of the tree. To this class we pass our MST as the graph and as the start vertex, we get a pseudo-random one by calling *next()* on our V iterator. As per the algorithm, the next step we take is adding a random edge connected to the root and we simulate that by calling *next()* on the BFSI and then asking the MST for the edge connecting them. That edge is the first one added to our matching. We keep track of the vertices we’ve used to compute the matching so we add the first edge’s vertices to this *verticesAdded* set. Why do we keep track of the ones we’ve used? That ties in with the next step. The next step is looping through all of the vertices to be visited by the BFSI where, for each iteration, we check if the vertex being examined is a leaf (by examining the number of edges connected to it) or not and act accordingly. If it is not a leaf, if none of its edges (and consequently it) have been added to the matching, we grab the set of edges connected to this vertex and iterate through it until we find an edge where the neighbor vertex has not been added to the matching where we then proceed to add the edge to the matching and record these vertices as having been used. Note that this allows a leaf to be part of the matching if the vertex used an edge connected to a child who is a leaf. If the vertex being examined is a leaf and it has not been used in the matching, we add it to our set of leaves.

One of the products of creating the matching and running the BFS is that we’re left with a set of leaves that have not been used in the matching. The next step in the algorithm is to orient these leaves towards their parent in the MST. This is where we actually start adding edges to our *TransmissionGraph*. This is pretty simple to do, although the math for figuring out the orientation angle can get a bit messy. In any case, if this leaf has an edge (it wouldn’t if the tree isn’t connected), we grab the vertex associated with the other edge, create a directed edge *from* the leaf *to* the parent and add it to the *TransmissionGraph* edge set. Then we have to orient the leaves so that the middle of their transmitted sector is aimed at the parent vertex.

The first step to finding the orientation angle is to find the height and stride (difference between the x-coordinates) difference *from* the vertex being oriented *to* the vertex being ‘pointed to’. The arctangent of this ratio will only give you a value between -90 and 90 degrees but depending on which quadrant we found the angle for (quadrant 1 has both delta height and delta stride positive, quadrant 2 has delta height positive and delta stride negative, quadrant 3 has both delta values negative and quadrant 4 has negative delta height and positive delta stride), the final orientation angle this represents is calculated differently. If we’re in quadrant 1, we can just use the calculated angle as the orientation angle; if we’re in quadrant 2, the orientation angle will be the calculated angle plus 180 degrees (since the angle is negative this gives us a result between 90 and 180 degrees); if we’re in quadrant 3, the orientation angle is the calculated angle plus 180 degrees; if we’re in quadrant 4, the orientation angle will be the calculated angle plus 360 degrees (since the angle is negative this gives us a result between 270 and 360 degrees).

The final step in the algorithm is to finish strongly connecting the MST. This is accomplished by: looping through every edge in the matching, orienting the two vertices of that edge so that they include their coupled neighbor at the extremity of their sector (done by setting the orientation to be the computed angle between the vertices as described above and then adding half of our sector angle to this result) and adding the two edges connecting this coupling to the TransmissionGraph edge set; then, looping through all of our vertices, if the vertex being examined is not one of the ones for the matching edge we’re looking at, if the non-coupled vertex lies in the sector area of either of the coupled vertex, add an edge to our TransmissionGraph going from the respective coupled vertex to the non-coupled vertex (this is where a leaf that wasn’t in our matching would get an edge connected to it). By the end of the iteration of the matching edges, if the graph was able to be strongly connected, it is oriented such that it is.

### Performance Comparisons

The shortest path comparison value represents an averaged ratio over the two graphs for the given trial. The way it’s computed is as follows: compute the set of all of the pairs in the power set of the set of vertices and call it P; for all sets S in P (they all have two sensors) get the hop count of the shortest path between the sensors in S for the Transmission Graph and the Proximity Graph; compute the ratio “Transmission Graph hop count” divided by “Proximity Graph hop count”; add this ratio to the result (which started at zero); once we’ve finished iterating through P, divide the result by the size of P.

The length of routes comparison value also represents an averaged ratio over the two graphs for the given trial. Its result is also initialized to zero and ends up being computed in parallel to the shortest path comparison value (i.e. during the same looping) as follows: for all sets S in P, compute the sum of the Euclidian distance of the shortest path between the sensors in S for the Transmission Graph and the Proximity Graph; compute the ratio “Transmission Graph route length” divided by “Proximity Graph route length”; add this ratio to the result; once we’ve finished iterating through P, divide the result by the size of P.

Computing the network diameter ratio is quite simple thanks to the JGraphT library. To compute the network diameter ratio we create two instances of the class *FloydWarshallShortestPath* (FWSP), one with the Transmission Graph passed into the constructor and one with the Proximity Graph passed into the constructor. We then iterate over the set of vertices and for each vertex, we call *getShortestPaths()* on both graphs’ instances of FWSP(which returns a list of all of the shortest paths from that vertex so we need to go through all of the vertices and combine the results). We then iterate over our lists of shortest paths and keep track of the sizes of the largest edge sets of each type of graph (the number of edges in the edge sets in this case represent the number of hops to travel between the two vertices in question and so the largest edge set count is the diameter of the network). Finally to get our ratio we divide the Transmission Graph diameter by the Proximity Graph diameter.

It should also be stated that computing the hop count and the shortest paths in these graphs were also done for us by the FWSP class.

## Testing

We didn’t do full-scale, professional testing on our program with formal test cases. Instead, as we implemented features we tested to make sure the new features worked and the old features were not broken. In that regard we did some kind of black box testing. The main testing to do was making sure the implemented orientation algorithm worked since that is what this assignment was really about. That being the case, we’ve come up with some basic cases to test against. Note that, thanks to JGraphT’s *StrongConnectivityInspector*, we are informed in the Graphical User Interface (GUI) whether or not our directed graph is strongly connected.

Table : Test 1

|  |  |
| --- | --- |
| **Objectives** | Determine whether or not our implementation’s results matched the behavior described in the class notes for directed antennae with a sector angle of 180 degrees and compare this to the omnidirectional version |
| **Expected Results** | We expected it to match and that is indeed what we saw |
| **Conditions** | The formed line has to be very close to being straight, sector angle 180 degrees |
| **Assumptions** | Same range and sector angle for all sensors |

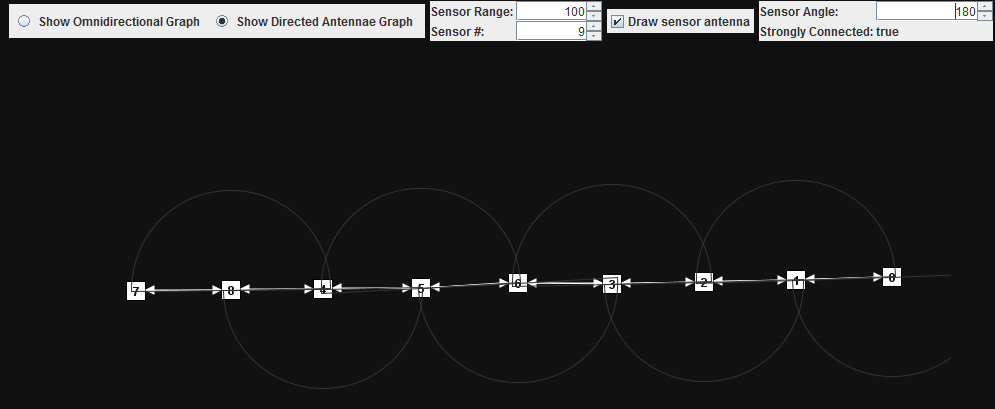


Figure : As we can see, when the sector angle is 180 degrees and they are aligned near perfectly straight, the algorithm orients the antennae such that they form a strongly connected graph

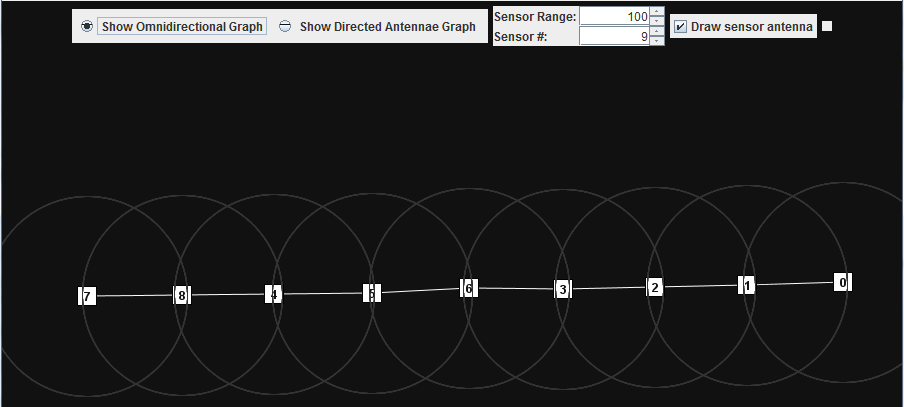


Figure : The omnidirectional version of the graph shown in Figure 1

Table : Test 2

|  |  |
| --- | --- |
| **Objectives** | Testing the algorithm on a more complex graph shape, ensuring proper antennae orientation without needing to change the sector angle |
| **Expected Results** | If there exists a placing of sensors into a complex shape such that the oriented graph is strongly connected, the algorithm will be able to properly orient the antennae and. When the sensors are spread out, the directed graph looks like the omnidirectional graph. Note that there are many placings that would result in a graph that is not strongly connected. |
| **Conditions** | The vertices have to be placed such that they should be strongly connected so the user needs to be able to figure that out and verify the result of the algorithm |
| **Assumptions** | Same range and sector angle for all sensors |

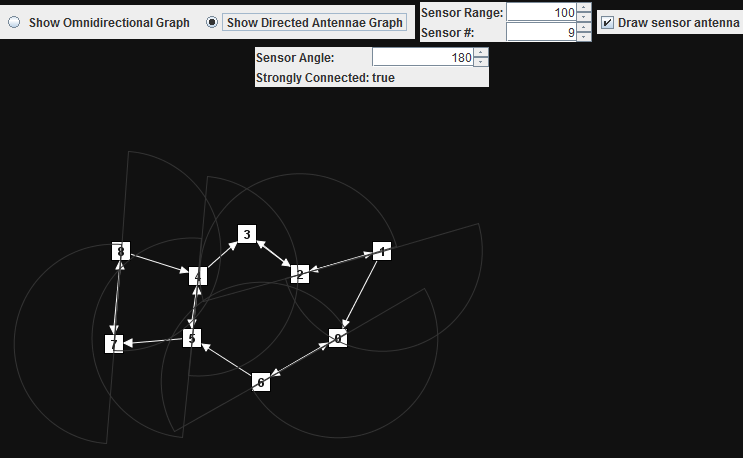


Figure : A manually, randomly shaped network using a 180 degree sector angle. Note that it is strongly connected. If we refer to Figure 4 we'll see that this is in fact a strongly connected graph of the omnidirectional counterpart.

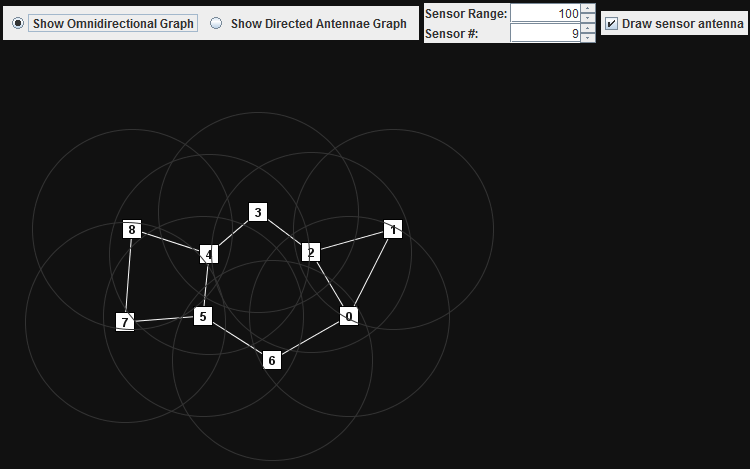


Figure : The omnidirectional graph of the network in Figure 3

Table : Test 3

|  |  |
| --- | --- |
| **Objectives** | Test the effects of altering the sector angle and range |
| **Expected Results** | Changing just one of these can connect or disconnect your directed graph and changing both may have the effect of preserving the connectivity status. We expect the algorithm to properly orient the sensors. |
| **Conditions** | The user needs to be able to figure out the connectivity of the graph and verify the result of the algorithm |
| **Assumptions** | Same range and sector angle for all sensors in a trial |

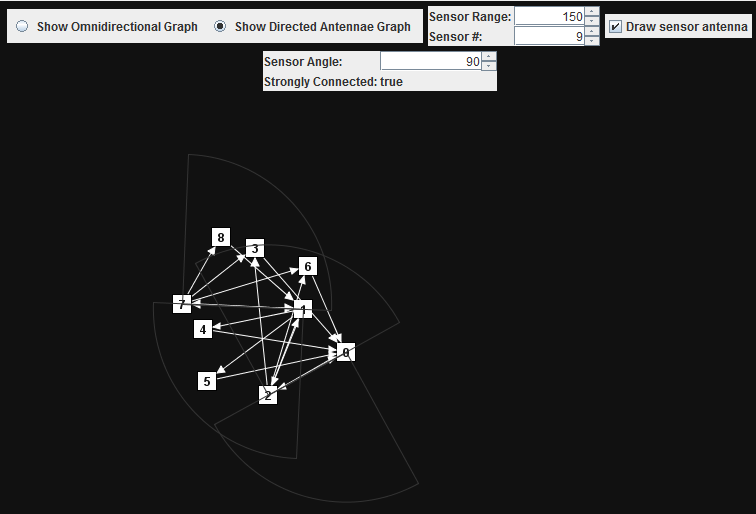


Figure : When contrasted to Figure 6 shows how a range change for a given sector angle can affect your connectivity. With a difference of 50 units (randomly chosen difference) these two graphs differ in connectivity.

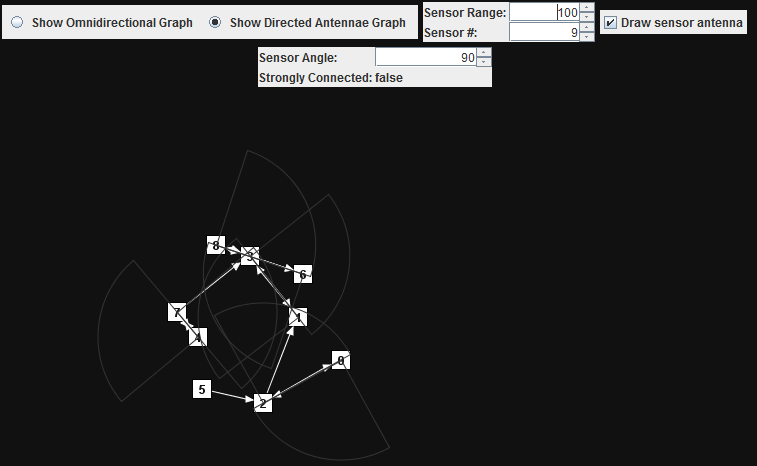


Figure : Refer to Figure 5

Table : Test 4

|  |  |
| --- | --- |
| **Objectives** | Test the algorithm output when the sector angle is 360 degrees |
| **Expected Results** | The algorithm should produce a graph that very closely resembles the omnidirectional counterpart when the sector angle is 360 degrees |
| **Conditions** | The sector angle is 360 degrees |
| **Assumptions** | Same range and sector angle for all sensors |

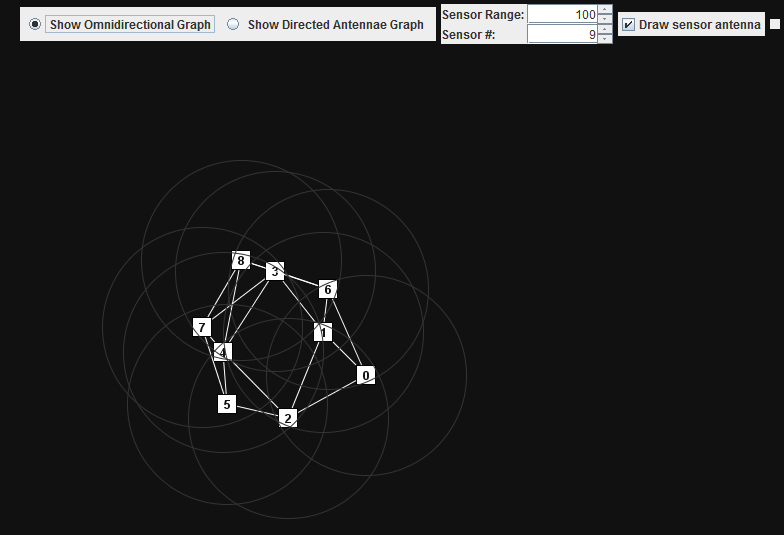


Figure : Another manually, randomly shaped omnidirectional graph. In some cases (this being one of them) the graph created by the orientation algorithm when its sector angle is 360 degrees (Figure 8) will have edges in the same spots as the omnidirection version which helps us validate the result of the algorithm.

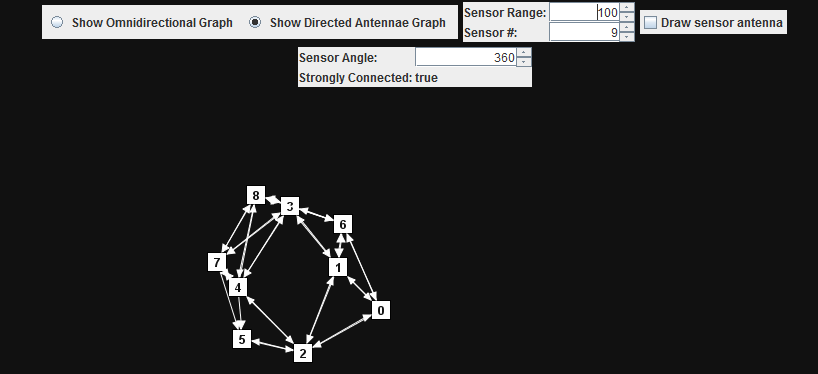


Figure : Refer to Figure 7

Table : Test 5

|  |  |
| --- | --- |
| **Objectives** | Explore a case where the sector angle is 360 degrees but the algorithm output is noticeably different than the omnidirectional counterpart |
| **Expected Results** | This does not mean the algorithm is broken. In fact, it is an indication of it working. See Figure 10’s caption for an explanation. |
| **Conditions** | Sector angle of 360 degrees |
| **Assumptions** | Same range and sector angle for all sensors |

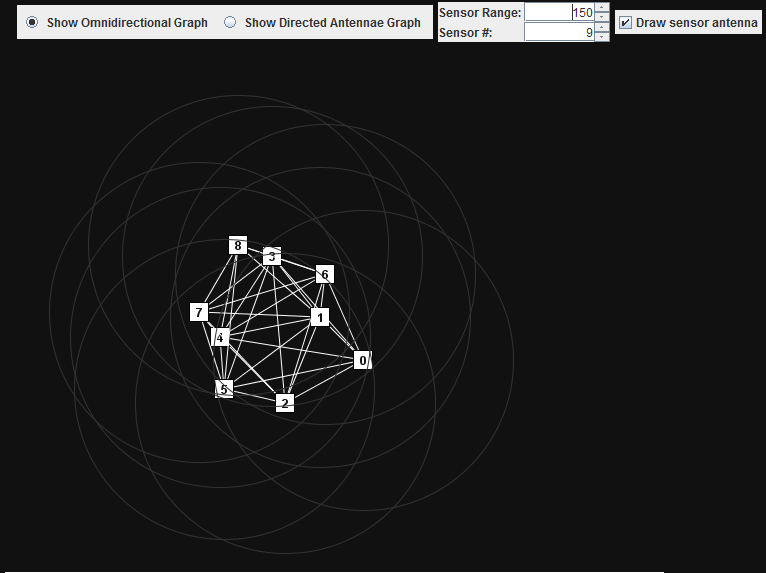


Figure : When viewed with Figure 10 goes to show that the edges in the omnidirectional graph don’t always get mapped to the directed version as was sort of the case in Figure 7 and Figure 8.

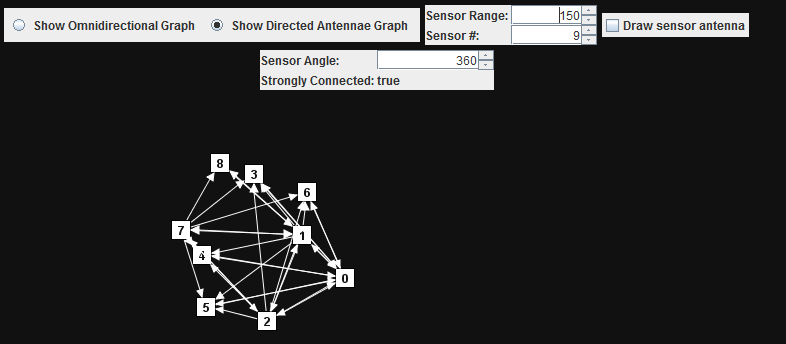


Figure : When we compare this graph to the one in Figure 9, although it is still strongly connected, it doesn’t have all of the same edges. This is a direct consequence of the algorithm and happens because the only vertices for which we make a connection to everything within reach are the vertices in the matching whereas the omnidirectional graph connects each vertex to everything it can reach.

## Logging

For each test we run, we log the data of the sensors in our vertex set, the individual/intermediate results of the shortest path and route length comparisons and finally the average shortest path ratio, average route length ratio and the average network diameter ratio.